

## **Field of the Invention**

## **Background of the Invention**

30 SSL does not support tunnelling in the sense of having an accessed-controlling intermediate system forward messages from a web browser to a target server. More particularly whilst a web proxy can be set up to act as a relay for SSL messages, there is no authentication of the browser by the web-proxy.

IPSEC is another known security protocol (see Reference [4]). IPSEC deals with network link-level security and therefore exists much lower down the network stack than SSL and addresses different problems. IPSEC supports identity-driven tunnelling. IPSEC tunnels treat all packets on the same "simplex" connection as being part of the same tunnel. The IPSEC headers contain indication whether they are being employed in transport or tunnel mode and this information may not be available in the Security Association Database; the receiving entity therefore has to work out for itself whether it is meant to be effecting a tunnelling operation.

It is an object of the present invention to provide a more security protocol layer for communication between remote applications that provides simple support for tunnelling through access-controlling intermediate systems.

#### **Summary of the Invention**

According to the present invention, there is provided a system with a local application entity and communications means by which the local application entity can exchange application messages with peer remote application entities on other systems, the communication means including a transport entity for providing transport services, and a security entity logically positioned above the transport entity and operative to set up secure communication sessions with peer security entities in other systems for the passing of application messages in protocol data units (PDUs) exchanged between the security entities, the security entity including a tunnelling mechanism for establishing a tunnel through an access-controlling intermediate system whereby to enable the local application entity to exchange application messages securely with a remote application entity on another system reachable via said intermediate system, the tunnelling mechanism establishing this tunnel by first setting up a first security session with said intermediate system and then a nested, second, security session with said another system with PDUs associated with the second session being encapsulated within PDUs associated with the first session and being extracted by the intermediate system for sending to said another system; and each PDU having a message-type field by which the security entity in the

intermediate system can determine whether a PDU it receives encapsulates a PDU to be extracted and sent on.

According to another aspect of the present invention, there is provided a system with a local application entity and communications means by which the local application entity can exchange application messages with peer remote application entities on other systems, the communication means including a transport entity for providing transport services, and a security entity logically positioned above the transport entity and operative to set up secure communication sessions with peer security entities in other systems for the passing of application messages in protocol data units (PDUs) exchanged between the security entities, each said security session being between specified application entities and the establishment of a security session being effected through a handshake process between the security entities concerned during which each application entity involved is required to show, by attribute certificates exchanged between the security entities, that it possesses certain attributes required of it by the other application entity; the security entity including a tunnelling mechanism for establishing a tunnel through an access-controlling intermediate system whereby to enable the local application entity to exchange application messages securely with a remote application entity on another system reachable via said intermediate system, the tunnelling mechanism establishing this tunnel by first setting up a first security session with said intermediate system and then a nested, second, security session with said another system with PDUs associated with the second session being encapsulated within PDUs associated with the first session and being extracted by the intermediate system for sending to said another system.

According to a further aspect of the present invention, there is provided an arrangement comprising first, second and third systems each with a respective application entity and communications means by which the application entity can exchange application messages with the application entities on the other systems, the communication means including a transport entity for providing transport services, and a security entity logically positioned above the transport entity and operative to set up secure communication sessions with the security entities in the other systems for the passing of application messages in protocol data units (PDUs) exchanged between the security entities, the

establishment of security sessions being dependent on each participating entity proving possession of, or access to, particular attributes, if any, required of it by the other entity; the security entity of the first system including a tunnelling mechanism for establishing a tunnel through the second system to the third system whereby to enable the application entity of the first system to exchange application messages securely with the application entity of the third system, the tunnelling mechanism establishing this tunnel by first setting up a first security session with the second system and then a nested, second, security session through the second system to the third system with PDUs associated with the second session being encapsulated within PDUs associated with the first session and being extracted by the security entity of the second system for sending to the third system; the application entity of said second system being a broker application that fronts for a target application constituted by the application entity of the third system, the application entity of the first system on wishing to contact said target application causing its security entity to seek to establish a security session with the broker application as said target application, and the broker application being operative to respond by causing the security entity of the second system to return to the security entity of the first system, an indication that the broker application is a relay for the target application entity, the application entity of the first system being thereupon operative to request a tunnel be set up through the broker system by the tunnelling mechanism, the attributes required of the second system by the first system being potentially different when the second system is providing a tunnel rather than hosting the target application entity.

#### **Brief Description of the Drawings**

A session level security protocol and entity embodying the invention will now be described, by way of non-limiting example, with reference to the accompanying diagrammatic drawings, in which:

- . **Figure 1** is a diagram of an end-to-end communication arrangement showing details of an embodiment of a Session layer Security (SLS) protocol entity used to establish a secure session over the communication arrangement;
- . **Figure 2** is a diagram depicting the basic Diffie-Hellman key exchange protocol;
- . **Figure 3** is a diagram showing the addition of signatures to the Figure 2 protocol;

**.Figure 5** is a diagram showing the addition of justifications to the Figure 4 protocol:

5     **. Figure 7**     is a diagram showing the addition of attribute advisories to the Figure 6 protocol;

**. Figure 9** is a diagram of the SLS key exchange handshake protocol:

. **Figure 11** is a diagram of a receiving subsystem of the secure channel functional block of the SLS entity of Figure 1;

**Figure 13** is a protocol stack including an XML generator layer below an SLS layer:

. **Figure 15** is a diagram illustrating the use of an SLS plug-in in a web browser for establishing a secure session with a resource mediation environment on a broker server; and

### Best Mode of Carrying Out the Invention

Figure 1 depicts an end-to-end secure communication path between a client 10 of a first end system 11 and a target service 12 of a second end-system 13 which client 10 wishes to use. This communication path involves a reliable connection 16 established between the end systems 11, 13 by transport entities 14, 15 of the two systems. The precise details of the transport mechanism used to establish and maintain connection 16 is not of importance to the present invention; however, by way of example, the connection 16 can be a TCP/IP connection. Typically, the transport entities 14, 15 are arranged to handle multiple

simultaneous connections potentially of different types, and this is represented by arrows 17 in Figure 1. Each connection, such as connection 16, may carry traffic for multiple simultaneous sessions of one or more applications (the client 10 being one such application) as is indicated by arrows 18. In the following The following description, which is primarily concerned with how security is provided during communication between two applications (client 10 and service 11), will focus on the provision of security for a single secure session between the client 10 and service 12 over the connection 16.

Security for communication between client 10 and service 11 is provided at the level of a session by cooperating session-level security ('SLS') entities 20, 30 in end systems 11, 13 respectively, the SLS entity 20 being logically located between the client 10 and transport entity 14 and the SLS entity 30 being logically located between service 12 and transport entity 15. Each SLS entity is capable of handling multiple simultaneous sessions involving different client-service pairings. The protocol operated between the SLS entities is herein referred to as the SLS protocol.

When the client 10 wishes to establish a communication session with service, the SLS entities first carry out a handshake procedure the purpose of which is two-fold:

- to determine if each party has certain 'attributes' required of it by the other - if this is not the case, then a communication session is not established
- to exchange cryptographic data to enable shared keys to be established for the communication session being established (if allowed).

Assuming the handshake was successful, the SLS entities are then responsible for operating the resultant secure channel established between the client 10 and service 11.

An 'attribute' expresses some property such as a name, a location, a credit limit, an owner or a role. Attributes are proved by certificates that are exchanged and authenticated to ensure that neither party (client or service) is making false claims about the possession of an attribute. Whilst the identity of the client or service constitutes an attribute, there is no *a priori* requirement that this attribute must be presented and verified - it is up to the parties (client 10, service10) to specify what attributes they require of the other.

In the present arrangement, attributes are established by SPKI certificates which are explained in detail in references [2] and [3]. It should be noted that as used herein, the term 'attribute certificate' means any signed electronic instrument bestowing an attribute on the subject of the certificate and therefore encompasses both SPKI 'attribute' certificates and SPKI 'authorization' certificates.

Proving that a party has a particular attribute means establishing a trust chain of valid certificates back to a party inherently trusted in relation to matters concerning the attribute. This trust chain may involve not only attribute certificates but also 'name' certificates. In this respect, it is useful to note that the issuer and subject of an SPKI certificate is fundamentally a principal constituted by a public key (or it's hash). Of course, there will be a keyholder associated with the public key (this being the party holding the private key matching the public key) but that party may not be identified at all. The subject of certificate may also be specified by a name but in this case there should also be a name certificate mapping the name to the corresponding principal.

A more detailed discussion of SPKI certificates and their use in proving attributes can be found in our co-pending UK patent application filed on the same date as the present application and entitled "Method and Apparatus for Discovering a Trust Chain Imparting a Required Attribute to a Subject" to which reference is directed.

In order to provide the means for implementing the foregoing features, the SLS entity 20 (and correspondingly the entity 30) comprises:

- a certificate services block 21 for providing trust chain discovery and certificate reduction services;
- a cryptographic services block 22 for providing signature creation and checking services and exponentiation services during the key-exchange handshake, key generation services for generating the session keys for the secure channel established following a successful handshake, and MAC (Message Authentication Code) creation/checking services and encryption/decryption services for message exchanged over the secure channel; and

- a protocol engine 23 with a key exchange handshake functional block 24, a secure channel functional block 25, a SLS PDU processing block 28, and a control block 26.

The control block 26 is responsible for coordinating the other elements of the protocol engine 23 according to input received from the client 10 and present in the unencrypted header fields of messages received over connection 16 via the transport entity 14. As already mentioned, the SLS entity is capable of handling multiple simultaneous sessions and the control block 26 is responsible for correctly associating client input and messages with the appropriate secure communication session (or to initiate a new session if no session currently exists when client 10 requests to communicate with services 12); this it does by assigning an identifier to each secure session, this identifier being herein called the Security Parameters Identifier (SPI) following the terminology used in the IPsec documents (see reference [4]). The SPI is carried in clear by messages passed over the secure channel. The control block 26 stores information relevant to each session, including the related SPI, in a session memory 27 and whenever the protocol engine receives a message from transport entity 14, it uses the SPI to look up the appropriate session data. The session data can also be accessed by client id. Block 29 in Figure 1 indicates the most important data items held for each session, the role of each item being explained hereinafter when describing the key-exchange handshake and the processing of messages for the secure channel.

The client 10 holds its own private key 33 used for digitally signing messages during the key exchange to authenticate them as originating from itself. The signing functionality is provided by the cryptographic services block 22. The client also holds its own collection of SPKI certificates in certificate library 32 from which it can extract the certificates appropriate for proving that it has particular attributes. Whilst the client endeavors to keep a record of the chain of certificates appropriate for proving each of its attributes, it can call on the trust chain discovery functionality provided by the certificate services block 21 to help it find such a chain (a suitable form of trust-chain discovery engine is described in our above-mentioned co-pending patent application). The client 10 can also call on the certificate services block to prove that a set of certificates provided by the service 12 actually prove that the latter has required attributes (proving this may require not only the certificate reduction functionality of block 21, but also the trust chain discovery functionality if the certificates are presented out



of order); the signature verification service of the cryptographic services block 22 will also be needed to verify that the certificates presented check out.

SLS is a layered protocol with a base layer implemented by the SLS PDU processor 28, the latter serving to assemble/disassemble SLS PDUs (protocol data units) by adding/removing a heading to/from a payload formed the output of the higher SLS protocol layer (for example the key-exchange protocol operated by the handshake protocol engine 24 or the secure channel protocol operated by the secure channel protocol engine 25). The general form of the SLS PDUs is depicted in Figure 1 for a PDU 35. The PDU 35 has, in addition to a payload 39, a heading made up of three fields as follows:

- a header field 36 containing the receiving party's SPI for the current session, to and from addresses (in any form suitable for transport entity 14), and a message serial number  $c$  described below;
- a message type field 37 indicating one of the following four message types:
  - HANDSHAKE
  - APPLICATION (payload to be passed to application)
  - TUNNEL (messages for nested sessions)
  - ALERT (error messages)
- an encoding type field 38 indicates the security processing in force as determined by the current cipher suite (see below), namely, clear text, a message protected by a MAC or an encrypted message (also with a MAC)

The key exchange handshake protocol effected by the handshake protocol engine 24 to set up a secure channel will next be described in detail, the approach taken being to build up the protocol element by element. In keeping with cryptographic tradition, the participants in the protocol are called 'Alice' and 'Bob' with Alice initiating the protocol exchange. Alice and Bob respectively correspond to the client and service sides of the Figure 1 arrangement - it should, however, be noted that Alice encompasses both the client 10 and SLS entity 20 and Bob encompasses both the service 12 and the SLS entity 30 though it will be apparent from what has been written above, where any particular processing takes place.

### Key Exchange Protocol

As will be seen, the key exchange protocol used in the present SLS implementation provides the following features:

1. Each party can present a set of attributes to the other party.
2. Each party is convinced that the other party has the claimed attributes if the protocol completes.
3. The protocol results in a shared secret key that can be used for establishing a secure channel. Only the two participants in the protocol have access to this key. The key is tied to the attributes that have been demonstrated by both parties.
4. The attribute-proving system is only required to be signature-capable. That is, it is assumed that the attribute certification scheme ties attributes to the possession of a signature-capable private key. The protocol uses only the signature attribute of this key.
5. If the private signature keys used in this protocol are compromised after the protocol has for use in the secure channel is not compromised.

Basic Diffie-Hellman Key Exchange Protocol. The protocol is based on the well known Diffie-Hellman key exchange protocol shown in Figure 2. Alice chooses a random number  $a$ , computes  $g^a$  and sends it to Bob in a handshake\_Start message hsS; Bob chooses his own random number  $b$ , computes  $g^b$  and sends this value to Alice in a handshake\_Reply message hsR.

The computations are done in a group  $G$ , here called the Diffie-Hellman group or DH-group for short. The element  $g$  is a generator of  $G$ . Both  $a$  and  $b$  are random numbers in the range  $0, \dots, r - 1$  where  $r$  is the order of the group  $G$ . The operator  $\in_R$  denotes the choosing a random element in a range.

The reader does not need to understand the mathematical concepts' of a group, its generator or its order. It is sufficient to know that a group is a structure which defines a multiplication. Exponentiation is defined by repeated multiplication, as it is done in the case of integers. Implementers will need to understand the group structure in order to implement the necessary multiplication. The actual group that is used can be varied; this is a choice that has its own

tradeoffs between message size, efficiency, security, and simplicity of code. Suitable groups can readily be specified by competent cryptographers.

The DH protocol is based on the fact that Alice can compute  $g^{ab}$  given  $a$  and  $g^b$ , and that Bob can compute the same value  $g^{ab}$  given  $b$  and  $g^a$ . However, an attacker that has access to the communicated messages knows only  $g^a$  and  $g^b$ . In a suitable DH-group it is computationally infeasible to compute  $g^{ab}$  given only  $g^a$  and  $g^b$ . The end result is therefore that both Alice and Bob have the same number  $k$ , but an eavesdropper cannot know  $k$ .

The secret key  $k$  can now be used to establish a secure link to exchange further messages. These messages can be authenticated and encrypted using  $k$ . This provides confidentiality of the message contents and proves to the receiver of the message that it originated from the sender.

DH with signatures The basic DH protocol has a weakness in that Alice and Bob only know that they are talking to someone, but not whether they are talking to each other. To resolve this problem two digital signatures are introduced into the protocol (see Figure 3) which require a private key to create and a public key to verify. The data that is signed in each signature consists of all the message data that was exchanged in the protocol before the signature. Thus  $SIG_B$  signs  $g^a$  and  $g^b$  in handshake\_Reply message hsR, and  $SIG_A$  signs  $g^a$ ,  $g^b$  and  $SIG_B$  in a new message, handshake\_Finish hsF. This signing of all previously-exchanged data is maintained for the protocol extensions discussed below where further data is added to the various messages (though, for clarity, not all data elements are necessarily shown in every depiction). As will be discussed below, the digital signatures are used in the handshake protocol as part of a process to establish that each party is the keyholder of the private key corresponding to the public key by which the signature can be verified; the identity of the keyholder, other than in terms of their public key, may or may not be thereafter verified by reference to a certificate.

From Alice's point of view the key exchange protocol so far establish looks as follows. She sends out  $g^a$  and receives  $g^b$  and  $SIG_b$  back. (The third message does not provide any new knowledge to Alice.) This allows her to draw several conclusions:

1. Somebody knows the secret key necessary to create  $SIG_b$ . Alice sent a random value  $g^a$  and received a signature back that included this value. This signature must therefore have been created after Alice chose  $a$ , which implies that somebody (which we call 'Bob') has access to that private key.
2. The pair  $(g^a, g^b)$  was approved by Bob. In this protocol the interpretation is along the lines of: "I received  $g^a$  and chose  $g^b$  as my reply in the DH protocol that we are executing."
3. The value  $k$  is only known to Alice and Bob.

Bob can derive much the same conclusions as Alice can. Alice (i.e. the party that knows the private key needed to create signatures) has approved the pair  $(g^a, g^b)$ , and  $k$  is only known to Bob and Alice.

DH Group Flexibility. As mentioned there are many possible DH groups that can be used. As time progresses and computers get faster it will become necessary to use larger and larger groups (or a different type of group) to maintain the same security level. It is therefore not possible to fix the group to be used. Furthermore, as mentioned, the choice of group has its own tradeoffs and different installations might choose differently.

Figure 4 shows how the protocol is extended to accommodate varying DH groups. Alice starts out by choosing a suitable DH group  $G$ . For most implementations only a single group will be used, and Alice will use whatever group was configured (generating a DH group usually takes a fair amount of time, so Alice should not do this on-line.) Alice sends the definition of the DH group to Bob in the first message  $hsS$ . The definition of  $G$  includes the type of group and the choice of the generator  $g$ .

When Bob receives the group definition he has to perform some checks to ensure this is a valid and properly chosen DH group. This is necessary as there are many valid group

definitions that are cryptographically weak. If Bob does not check the group then he cannot later trust the secure channel that this protocol sets up.

Note that the group definition is included in both signatures. This ensures that both parties are using the same group and that an attacker cannot manipulate the parties into using different groups in a single protocol execution.

Checking the validity of a group typically involves at least an order of magnitude more work than the rest of the protocol. To avoid having to perform the checks every time, Bob preferably keep a cache of 'approved' groups that have passed the checks.

Adding Certificates. As already indicated, in the SLS protocol, a major purpose of the key exchange protocol is to verify attributes. That is, Alice wants to verify that Bob has certain attributes and Bob wants to verify that Alice has certain attributes. These attributes can express an identity ('I am Bob'), a property ('I work for HP labs in Bristol'), or an authorisation ('I am allowed to use the printers on floor 2').

The key exchange process has a fairly simple interface to the attribute system. For party to prove that it has a attribute two things are needed: a attribute justification linking the attribute with a public key (allegedly that of the party concerned), and a proof that the party possesses the related private key. The attribute justification will typically consist of a set of certificates from the certificate library 32 of one party and verified by the attribute justification system of the other party. A justification implies a statement of the form: 'The holder of the private key corresponding to the public key  $X$  has attributes  $c_1, c_2, \dots$ '.

The extension of the protocol as so far described to accommodate Bob and Alice proving to the other that they possess certain attributes involves each of them sending their own attribute justification JUST, and using their corresponding private key to create the signature already in the protocol (note that Alice or Bob may well have more than one associated public/private key pair with certain attributes being associated with one public key and other attributes with a different public key - obviously, the correct private key must be used for the attribute

requiring to be proved). The exchange of attribute justifications is shown in Figure 5: Bob's attribute justification  $JUST_B$  is included in message  $hsR$  whilst Alice's attribute justification  $JUST_A$  is included in message  $hsF$ . Bob's signature is done with the key that proves possession of the attributes that are the subject of attribute justification  $JUST_B$ . Similarly,

5 Alice's signature is done with the key that proves possession of the attributes according to  $JUST_A$ .

As previously indicated, the protocol does not necessarily identify the two parties other than by their public keys.

10 Attributes that depend on multiple keys. There may be some situations in which it is simpler for the attribute system to use several private keys. In that case the justification corresponds to a statement of the form: 'The holder of the private keys corresponding to the public keys  $X_1, X_2, \dots$  has attributes  $c_1, c_2, \dots$ '.

15 The protocol can support this - instead of sending a single signature, the party that uses such a multi-key justification sends a separate signature with each of the private keys. For example, if Bob uses such a justification,  $SIG_B$  would consist of a list of signatures, one for each of the keys required by the justification. For simplicity, all these signatures sign the same data string

20 which consists of all the data exchanged before the signatures. (Thus, the last signature does not include the earlier signatures of the same list in its signature data.) For simplicity, this generalization is ignored for the remainder of this discussion of the key exchange protocol as to include it makes the overall description more complex without adding any significant understanding. If required, multi-key justifications and signatures can be used in the SLS

25 protocol.

Secure channel identifiers As already mentioned, individual sessions are identified by an SPI. The SPI assigned by Alice to the session with Alice will generally be different of that assigned by Bob to the same session but this does not matter. By having Alice and Bob

30 exchange the SPI's they have assigned to the session, each can attach to any message it sends the other the SPI assigned by the latter. The advantage of letting each party choose their own

SPI is that some or all of the bits of the SPI can be used directly as an index into a table. This speeds up processing on the receiving side.

Figure 6 shows how the  $SPI_A$  and  $SPI_B$  fields are added to the messages,  $SPI_B$  being included in message  $hsR$  and  $SPI_A$  being included in message  $hsF$ . These fields are automatically included in the signatures. At the end of the protocol Alice and Bob store the SPI values with the other information of the secure channel for identification purposes.

Attribute Selection Advice. During the protocol, Alice and Bob have to provide a set of attributes. This poses the question how they know which attributes to include. It is certainly inadvisable to include every attribute they possess. A system administrator might easily have thousands of attributes. To deal with this problem, a system of attribute advisories is used to help the participants select the appropriate attributes.

The attribute advisory system is asymmetric; that is, the features it provides to Alice and to Bob are slightly different. There are two reasons for this asymmetry. First of all, the 3-message protocol is inherently asymmetrical and adding advisories results in an asymmetric advisory system. The advisory system could be made symmetrical by reducing the functionality for one side, or by adding two messages to the protocol. Adding messages is too expensive (in terms of additional delay, and complexity). Restricting the functionality would force the system to discover the information in a different manner, probably using at least two additional messages.

The second reason is that the requirements are actually different for the two parties. In this protocol Alice is the initiator. She is the one who wants to send a message to Bob and set up a secure channel for it. Presumably Alice knows what she intends to do, and which attributes she would like Bob to show. Bob, on the other hand, has no idea what Alice wants. Therefore, it can be impossible for Bob to give any good advise on what attributes he will require of Alice without some information about what Alice intends to do. The existing 3-message structure allows the parties to exchange exactly that information without adding additional messages.

A typical example would be that the client Alice tries to access the resource Bob. Alice knows what she wants to do with the resource, and which attributes she requires the resource to have. Bob has no idea what functions Alice wants to call on the resource.

5

The attribute advice system consists of three fields, as shown in Figure 7. The first field is in the first message  $hsS$ , and is called  $ADV_B$ . This is Alice's advice as to which attributes Bob should include in  $JUST_B$ . The other two fields are a more complex negotiation to help Alice determine which attributes she should include in  $JUST_A$ . In the first message  $hsS$ , Alice sends  $QRY_A$  to Bob. This is a query field which Alice and Bob are arranged to interpret in the same way.  $QRY_A$  contains the enough information for Bob to give a sensible  $ADV_A$  to Alice. In a client-resource negotiation, the  $QRY_A$  could consist of a set of resource operations Alice plans to call in the future. Bob can then inform Alice in  $ADV_A$  what attributes are required for this. The  $ADV_A$  field is included in the second message  $hsR$ .

10

15

Under most circumstances parties should satisfy the attributes listed in the advisory, but this is not mandatory - the advisories are just that, advisory. If Alice requests Bob to show attribute  $X$ , and he does not, then Alice is likely to refuse to use the resulting secure channel that the protocol establishes. Similarly, Bob is likely to refuse to process messages from the secure channel if Alice has not shown enough attributes during the key exchange protocol. In fact, either Bob or Alice is likely to decide not to complete the setting up of a secure channel if the other cannot show the required attributes.

20

25

There are several situations in which a party would want to send more attributes than those that are requested directly. For example, if a client opens a file for reading, but knows she will want to write the file later, then it is more efficient to include enough attributes in the protocol so that the secure channel can also be used to write the file. This clearly indicates that a party should be free to include more attributes than those requested.

30

It could be argued that it never makes sense to include fewer attributes than those requested. But there is a significant functional difference between the protocol failing with an error, and



the recipient of the message sending an error reply which includes error codes and other information. There are also situations in which a party wants to continue even though it does not have enough attributes. For example, Alice might want to access a high-speed network if it is available and not being used for video-conferences. If the high-speed network is not available, she knows she has access to the low speed network.

If Bob is merely fronting for a service Charlie, then when Bob is contacted by Alice he is likely to be unable to satisfy Alice's requirements which Alice would have specified in respect of the service she wanted -the service actually offered by Charlie. As will be described below, Bob can indicate this by setting a "relay" flag in the hsR message and it is then up to Alice to decide whether she wishes to access Charlie by tunneling through Bob.

Choice of Cryptographic Algorithm. There remains the issue of how Alice and Bob choose the cryptographic algorithms to use. First, considering the cryptographic functions used in the key exchange protocol itself, it will be recalled that selection of the DH group has already been made flexible with the group chosen by Alice being communicated to Bob. As regards the digital signatures, the various public keys and signatures all use their own algorithm so that if, for example, Alice has a DSS signature key to sign  $SIG_A$ , then Bob has to know how to verify a DSS signature. However, it is assumed that all public keys and signatures are encoded in such a way that their type can be recognised by the recipient and that the recipient will be able to handle at least the most common types.

With regard to the cryptographic algorithms to be used for the secure channel once established, there are many possible choices (though generally symmetric key algorithms will be used for encryption/decryption as they are much faster than asymmetrical algorithms). The solution here employed for choosing which algorithms to use is similar that adopted in SSL 3.0 and involves centrally defining cipher suites where each cipher suite is a full set of cryptographic algorithms for the secure channel. Each cipher suite is assigned a unique identifier by a central authority. During the key exchange protocol Alice and Bob negotiate a suite to use. Alice sends a list of suites she is willing to use, in decreasing order of preference. Bob selects one of the suites on the list and informs Alice of his choice. This is the suite that both will use.

Figure 8 shows the necessary addition to the protocol thus far built. Alice chooses a list of cipher suite identifiers  $CS_L$ . This list is in decreasing order of preference. This list is sent in the first message  $hsS$  to Bob. When Bob receives the list he selects one of the cipher suites.

5 Bob should choose the first cipher suite in the list that is acceptable for him. This is the cipher suite that will be used for the secure connection. Bob sends his selection  $CS$  in the second message  $hsR$  to Alice, who in turn checks that this selection is valid. An example of one possible combination of algorithms making up a cipher suite is given hereinafter when describing the secure channel.

10 Final Handshake Protocol The final key exchange handshake protocol is shown in Figure 9. This is identical to that depicted in Figure 8 except for the added operation of storing the information required for the secure channel (for Figure 1, this would be in session memory 27). This information is the result of the protocol.

15 Alice would typically store a communication address that will reach Bob, the list of attributes that Bob showed to Alice in the protocol, the list of attributes that Alice showed to Bob during the protocol, the validity periods and conditions of the various attributes, and the shared secret key  $k$ . Bob would store similar information.

20 It will be appreciated that the handshake messages are incorporated into PDUs by SLS PDU processor 28 before transmission, the message type of such PDUs being set to HANDSHAKE.

25 The key exchange protocol was initiated by Alice to be able to send a message to Bob. Note that Alice has all the information about the secure channel before she sends the last message  $hsF$  in the protocol. It is therefore possible for Alice to send the first message on the secure channel in the same PDU as the last message of the key exchange protocol. This is a preferred implementation as it can result in a significant overhead reduction; this requires the

30 use an appropriate message format that allows the last message of the key exchange protocol

to co-exist with the first message on the secure channel (the design of such a format is within the competence of persons skilled in the relevant art).

Application Identification Field. With respect to the signing of messages in the foregoing key exchange protocol, it should be remembered that what is being signed is a string of bytes; it is important that this string is parsable back into the original fields in a consistent manner. It is also important to know what the recovered fields mean and this depends on the application, the protocol that is being executed, and even the message within the protocol in which the signature is contained. Sometimes a signature is used for identification, sometimes it is used for authentication. A signature can imply a legally binding contract, or just an approval for the message to traverse the firewall. Preferably, therefore, every message that is signed starts with an identification field that uniquely identifies the application in which it is used, and its function within the application. The application can be identified with its application name and version number. The function of a signature within an application is usually identified by indicating the protocol within the application in which it is used and the message within this protocol in which it is used.

Where such an initial identification field is used, the parsing rule for this field must be the same across all applications that use the same key. For example, the following rules can be used for this identification field.

1. The field is an ASCII string of characters without any control sequences.
2. The string consists of the application name, protocol name, protocol version, and name of the signature within the protocol. (Applications that store signatures on disk will, of course, require a different identification scheme but should still use a string that starts with the application name to avoid collisions).
3. Each string of bytes that is signed starts with the length of the identification string encoded as a 2-byte integer in network order (MSByte first), followed by the identification string in question.

As an example, consider the signatures in the above-described key exchange protocol. A suitable identification string for  $SIG_B$  would be "Hewlett-Packard/e-speak; client-resource key exchange protocol, version  $x/$  date; sigB". This identifies the application, the protocol and version number, and the function within that protocol. (The date is included in the version information to ensure uniqueness.) The identification string for  $SIG_A$  could be similar: "Hewlett-Packard/e-speak; client-resource key exchange protocol, version  $x/$ date; sigA". Note that the overhead of this identification field is very low. The application simply starts every string to be signed or verified with a fixed sequence of bytes (which include the 2-byte length encoding).

### The Secure Channel

The key exchange handshake protocol described above is performed in order to establish a secure channel between Alice and Bob. As will be seen, this secure channel has the following features:

1. Messages on the secure channel are authenticated. Alice will only accept messages that were sent by Bob and vice versa.
2. Messages are encrypted: an eavesdropper cannot determine the contents of the message being sent.
3. The sequence of message that Alice receives is a subsequence of the messages that Bob sent, and vice versa. This prevents replay attacks, reflection attacks, etc. Note that the channel does not guarantee that every message will be delivered, only that they will not arrive out of order.

The encryption and authentication functions of the secure channel use symmetric key cryptography and are therefore orders of magnitude faster than public-key functions.

The secure channel has a limited life-time. The attributes shown in the key exchange protocol typically have a limited validity period. The validity periods of the attributes exchanged are held in the session memory and checked regularly by the control block 26 (for example, before each message is processed. When these attributes expire the key exchange protocol will have to be re-run to set up a new secure channel. As a general rule, either party is free to discard the information relating to a secure channel at any time. This could, for example, be

done when the first attribute associated with the channel lapses, when there has been a significant period of time without any activity on the channel, or when the application signals that it will no longer need the channel. Preferably, when the protocol engine 23 decides to discard a channel, it sends a corresponding message to the other side of the channel; this message should of course be authenticated in the same way as all other messages.

To ensure proper error handling, the SPI numbers should not be recycled too rapidly by the protocol engine 23. If a message is received for an unknown SPI this will be signaled as such. If the SPI has been re-used, the message will generate an authentication-failure error, which does not provide the sender with the information that the old channel has been discarded.

The secure channel makes use of four functions in the selected cipher suite, namely a hash function  $h()$ , a MAC function  $MAC()$ , an encryption function  $E()$  and the associated decryption function  $D()$ ; the algorithms used for each function is determined by the cipher suite chosen for use. One possible cipher suite comprises:

- Hash function: - SHA-1 (see Reference [5])
- MAC function - HMAC-SHA1-96 (see Reference [6])
- Encryption / Decryption - 3DES in CBC mode (see References [7],[8])

Deriving Key Materials. The MAC function has two associated authentication keys and the encryption/decryption functions have two associated encryption (and decryption) keys. For deriving the authentication and encryption keys it is convenient to define a further function, namely a helper function  $h'$  that is used in conjunction with the hash function. More particularly, the hash function  $h$  maps arbitrary length input strings to a output strings of a fixed length. The length of the output string is fixed for any function  $h$ , but it depends on the choice of that function. The helper function  $h'$  provides a variable-length output and is part of the services provided by the cryptographic services block 22.

Given an input string  $s$  and a desired length  $l$ ,  $h'(s, l)$  is defined as follows:

$$s_0 := h(s)$$

$$s_i := h(s_0 | s_1 | \dots | s_{i-1} | s) \quad i \geq 1$$

$h'(s, l) :=$  The first  $l$  bytes of  $s_0|s_1|s_2|....$

where  $|$  is used to denote concatenation of strings. The hash function  $h$  is used repeatedly to generate enough output, and then take the  $l$  leading bytes are taken.

- 5 The construction has been chosen to ensure that the entropy of the result is not limited by the output size of the hash function or by the internal state size for a standard iterative hash function.

10 The encryption algorithm from the chosen cipher suite specifies the length of the encryption keys. The MAC algorithm from the chosen cipher suite specifies the length of the authentication keys.

The result of the key exchange protocol is a shared group element  $k$ . This must first be converted to a string using a canonical binary representation. The group definition should define such a canonical representation. We will call the canonical representation  $k_s$ .

15

The encryption keys are called  $K_{AB}^E$  for the key that will encrypt traffic from Alice to Bob and  $K_{BA}^E$  for the key that will encrypt traffic from Bob to Alice. Similarly the authentication keys will be called  $K_{AB}^A$  and  $K_{BA}^A$ . Let  $l_A$  be the length of the authentication keys and  $l_E$  the length of the encryption keys (as specified by the respective authentication and encryption algorithms). We now define

20

$$K_{AB}^A := h'(\text{"Authenticate Alice to Bob"} | k_s, l_A)$$

$$K_{BA}^A := h'(\text{"Authenticate Bob to Alice"} | k_s, l_A)$$

$$K_{AB}^E := h'(\text{"Encrypt Alice to Bob"} | k_s, l_E)$$

$$K_{BA}^E := h'(\text{"Encrypt Bob to Alice"} | k_s, l_E)$$

25

where the literal strings are plain ASCII strings (without any terminating zeroes).

Typically, both Alice and Bob will store all four keys in their session memories 27.

30

Sequence Counters. One generic attack against a protocol is to resend an old message. If the message is only authenticated then it will be accepted as real by the recipient. There are various ways in which replay attacks can be avoided. Here a sequence counter is used.

Alice keeps a sequence counter  $c_{AB}$  for the messages that she sends to Bob over the secure channel. This counter starts at 1 for the first message and increments for every message that Alice sends. Bob keeps track of the  $c'_{AB}$ , namely, the largest counter value he has seen in a proper message. If a new message arrives, and the MAC code checks out, Bob compares the counter value  $c$  in the message with  $c'_{AB}$ . If  $c \leq c'_{AB}$  then the packet is at least as old as the last accepted message, and it is discarded. If  $c > c'_{AB}$  then this is a newer message and the variable  $c'_{AB}$  is set to the value  $c$ . It is easy to see that this rule prevents Bob from ever accepting the same message twice, or accepting messages out of order. Of course, for messages from Bob to Alice the same mechanism is used with Bob keeping a counter  $c_{BA}$  and Alice keeping the largest valid counter value yet  $c'_{BA}$ .

Note that the first counter value that is sent in a message is 1. This allows the recipient to initialize the  $c'$  variable to 0 and proceed as normally.

It is generally good practice to limit the amount of data that is encrypted or authenticated with a single key. For example, for a 64-bit block cipher the number of blocks encrypted with any single key should not exceed  $2^{32}$ . This implies that the range of the sequence counters can be limited, for example to 32-bit unsigned integers. This allows  $2^{32} - 1$  messages to be sent on the channel. If this limit is ever reached, the secure channel has to be abandoned and a new channel negotiated using the key exchange protocol. Note that both Alice and Bob have to ensure that the counter values do not wrap around. Either party can also impose their own more strict limit and abandon the secure channel earlier. A new channel can always be negotiated.

Advantageously, the sequence number  $c$  of a message is provided to both the encryption (and decryption) block and the MAC block (in the case of the latter, this is in addition to  $c$  being included in the data in respect of which a MAC is to be formed). This is done to enable the use of certain encryption and MAC functions (if agreed by the parties) which have the potential of being faster. For example, if a stream cipher is used, care has to be taken that

every message is encrypted with its own unique key stream. The counter value allows the stream cipher to ensure that this is the case. It is also possible to have MAC functions that are very fast, but which can only be used once for each key. The counter value  $c$  can be used to derive unique authentication keys for each message from the channel authentication key.

5

Message Transform by Sender The processing of a message  $m$  at the sending end of the secure channel will now be described with reference to Figure 10 which depicts the main functional processing blocks of the secure-channel sending sub-system 40 and the message transform they each effect. As well as the message  $m$  the sending sub-system is supplied with a reference enabling it to retrieve the relevant session data (block 41); this data includes the sending sequence number  $c$ , the cipher suite and session keys to be used, and the SPI value for the recipient (thus Alice will use  $SPI_B$  for messages she sends to Bob). The message  $m$  is usually variable-length, while  $c$  and SPI are fixed length (here assumed as 4 bytes).

10

First,  $MAC := MAC(K_A, c, SPI || c || m)$  is computed (that is, the MAC function is applied to the string consisting of the concatenation of SPI,  $c$ , and  $m$  - see block 42). The appropriate authentication key  $K_A$  and the counter value  $c$  are also inputs to the MAC function.

15

Encryption is effected next (block 43) by computing  $q := E(K^E, c, m || MAC)$ ; in other words, the message  $m$  and the MAC field are concatenated and then encrypted using the appropriate encryption key  $K^E$  (with the counter  $c$  being an additional input to the encryption function as already explained). The result of the encryption is called  $q$ . There is no direct relation between the length of  $m$ , MAC, and  $q$ . Many encryption algorithms require padding, which increases the length, and some might implement compression which could reduce the length.

Note that here the padding is considered to be part of the encryption function as the padding requirements depend on the details of the encryption function being used. Note that the encryption step can result in an increase in overall message length.

20

25

30

The message (PDU) that is finally sent consists of the concatenation of SPI,  $c$ , and  $q$  plus certain additional information, such as addresses, as already described in relation to the fields

09733475 130700



The overall operation of the sending subsystem is coordinated by a state machine 44.

The received SPI is used by the receiving sub-system to find its internally-held information about the secure channel in question (block 51). Since the SPI has not yet authenticated, the sub-system needs to be robust and not crash even if the SPI is corrupted. For this reason, the SPI value should not be chosen as the memory address of a secure channel data structure holding the required information about the channel; the SPI value can, however, be chosen as an array index provided appropriate range checks are implemented during the lookup.

The value of the MAC field is then checked by re-computing the appropriate MAC value from the SPI,  $c$  and  $m$  field using the authentication key  $K^A$  (block 54) and comparing it with

the value in the MAC field (block 55). If these two values are different, the message does not pass the authentication check and is discarded.

Once the message has been authenticated the receiving sub-system performs its checks on the counter value (block 52), and updates its own  $c'$  value. If desired, the check that  $c$  is greater than the current  $c'$  can be done before decryption and authentication. However, updating the  $c'$  variable must only be done after authentication has been completed.

The overall operation of the receiving subsystem is coordinated by a state machine 56.

### Tunneling

The SLS protocol supports tunneling, that is, the passing of PDUs through an access-controlling intermediate system to a final destination. PDUs that are to be tunneled are encapsulated in SLS PDUs which have their message type (field 37, Figure 1) set to TUNNEL. Tunneling requires the consent of the intermediate system concerned as will be more fully explained below with reference to Figure 12.

As before, suppose the parties to the SLS handshake are Alice and Bob, with Alice initiating as client. When Alice sends a handshakeStart message to Bob she is expecting a handshakeReply from Bob that includes Bob's proof of the attributes Alice required of him. However, sometimes Bob is not in a position to supply the proof - for example, consider the case where Alice has the address of a service that appears to reside at Bob, but Bob is in fact a mediator (gateway application) for the service and forwards requests to another party, Charlie, who implements the service. Charlie is shown in Figure 12 as a system composed of a transport entity 61, an SLS entity 62 and a service 63. If Bob has no security restrictions of his own he can simply forward messages unchanged in both directions, and Alice and Charlie can set up a SLS session. If necessary, Bob can rewrite the to and from addresses of the messages to get them delivered to the right place. This is because the 'to' and 'from' addresses are not protected by the handshake signatures or MACs (they are in the header field 36). Everything else is protected.

Assume now that Charlie has allowed Bob to broker the service provided by Charlie and also to impose his own access restrictions. This means that Bob wants to check Alice's attributes before forwarding messages to Charlie. In order to check Alice's attributes Bob has to set up a SLS session with her. But since Bob is not the real service he may well be unable to prove to Alice the properties she requires of the service. This possibility is provided for in the SLS protocol by including a 'relay' flag in the handshakeReply message hsR sent by Bob to Alice.

When the relay flag is true, Bob is telling Alice that he is a mediator, and so may not be able to prove all Alice's required attributes. It is up to Alice whether this is acceptable. If it is, she can complete the handshake and set up a session with Bob (the Alice-Bob session 64). Alice now needs to set up a session with the entity Bob is relaying to – Charlie in this case (the Alice-Charlie session 65). Alice does this using Alice-Bob session PDUs of message type TUNNEL. These PDUs carry, as payload, PDUs for the Alice-Charlie session 65 (effectively a nested session within the Alice-Bob session). The PDUs for the Alice-Charlie session contain the messages (initially, handshake messages but subsequently encrypted message data), and the unencrypted PDU fields 36-38 - since this information will be visible as such on the Bob-Charlie connection, there is no great benefit in Alice encrypting the payload of the Alice-Bob session PDUs and this step can therefore be omitted to reduce processing overhead though forming a MAC for this payload should still be done. When Bob receives an Alice-Bob session PDU with its message type set to TUNNEL, he forwards its payload as a PDU to the mediated entity (Charlie). Bob performs the security processing negotiated for his session with Alice in the usual way. If Bob receives a PDU from Alice with message type set to APPLICATION rather than TUNNEL, Bob assumes the message is for him and attempts to decrypt the payload of the PDU in the usual way.

Alice now sets up the Alice-Charlie session 65 with Charlie. Notice that once a secure channel has been set up between Alice and Charlie, then assuming Alice encrypts the payload of the Alice-Charlie session PDUs, Bob will not be able to read the payload being passed to Charlie. All he will be able to see is the header fields 36 - 38.

In controlling tunneling, the control block 26 of the protocol engine 23 of Alice's system (the sending system) needs to keep a track of the session nesting. This can be done by including in

the data stored for each session the SPI of any immediately-nested session. Thus for the Figure 12 example, the session data for the Alice-Bob session 64 (which would generally be the session data initially retrieved by control block 26 when Alice indicates she want to send a message to Charlie) would show that the session was with Bob, not Charlie, and that there was a nested session with a stated SPI (being the SPI of the Alice -Charlie session 65). This tells the control block 26 that when sending data to Charlie the Alice-Bob session 64 is simply acting as a channel for a nested session with the consequence that the PDUs of session 64 should be set to type TUNNEL (the question of whether or not the payload of these PDUs is to be encrypted can be set by policy, by choice of cipher suite, or by an explicit flag set in the session data). The control block 26 next looks at the data for the session corresponding to the SPI in the Alice-Bob session data, namely the Alice-Charlie session data; this indicates that the receiver is Charlie (the required recipient) so that the PDUs for this session will have a message type APPLICATION and the payload will be encrypted.

If Alice wants to send a message to Bob, then when the control block looks up the session data for session 64, it can see that the recipient is Bob and so PDUs can be set directly to APPLICATION and there is no need to use any nested sessions; in other words, the control block does not need to concern itself with the fact that session 64 carries a nested session as indicated by the session 65 SPI in the session 64 data.

As already indicated, handling of tunneling by the recipient (Bob) of a PDU of the message type TUNNEL is very simple and does not require any tracking mechanism - the recipient simply take the payload of the received PDU, carries out any MAC-based checking (and possibly decryption) needed and forwards the payload (minus MAC, if present) to the entity (Charlie) indicated in the "to" address included in the header field 36 that forms part of the payload of the received PDU. Of course, the address of this entity is probably not known to Alice and Bob must supply this address, inserting it in the "to" address of the PDU to be forwarded. The address of Charlie is conveniently held by Bob in his session data for the Alice-Bob session ready for use when a TUNNEL PDU is received from Alice. Bob will generally also set the "from" address to show that the message is from him.

Although the forgoing example of tunneling concerned only two nested sessions, any number

of nested sessions are possible.

### **SLS Using XML**

An XML DTD (Document Type Declaration) for a preferred structure of an SLS PDU is given in the Appendix to be found at the end of this description. XML is a markup language details of which can be found in Reference [9]. An XML DTD describes the structure of a document and may be appended at the beginning of the document concerned. XML is finding rapid acceptance as a markup language for World Wide Web documents as it permits much more richly structured documents than is possible with HTML. As a result, many web browsers are now XML capable.

By defining an XML DTD for SLS PDUs, it becomes possible to effect the SLS protocol exchanges as XML documents, the latter being transportable over a wide range of transports including HTTP and e-mail. Figure 13 depicts an end system in which an SLS layer provides secure session services to an application.; interposed between the SLS layer and the transport layer is an XML generator layer 72. The transport layer comprises a set of transports 73 - 76 (including TCP transport 73, e-mail transport 74 and HTTP transport 76). The XML generator layer converts the SLS PDUs into XML documents that can then be transported by any of the indicated transports (among others). At the receiving system, an XML interpreter reforms the PDU into the form familiar to the SLS layer.

It is possible to use another self-describing markup language in place of XML including SGML (Standard Generalized Markup Language) though only larger systems tend to provide support for this latter markup language.

### **"E-Speak" and SLS**

An intended application of the above-described SLS protocol is in providing security to Hewlett-Packard's "E-speak" technology. It is useful in understanding the capabilities of the SLS protocol to consider examples of how the protocol can be deployed with such technology. A brief overview of the E-speak technology is given below to aid an understanding of the SLS deployment; a more detailed exposition of the technology can be found at reference [10].

E-speak deals in terms of "resources" described by metadata. The metadata defines resource type, attributes and security properties. Resource metadata is registered with a repository in an e-speak daemon known as a "core"; this metadata can be exported from core to core. An active service registers itself as the handler for a resource with a core which then forwards messages for the resource to the handler. For present purposes, the handler and resource will be treated as equivalent to a "service" application such as the service 15 of Figure 1; also, for simplicity, the resource and handler will not be distinguished and are jointly referred to below as a "resource".

A client typically connects to a core, does an attribute-based lookup to find a resource and then is able to invoke the resource. The core passes messages from the client to the resource. All resources are referred to by name.

The SLS layer is intended to form a part of an e-speak core to provide security (access control and confidentiality) for communication between a client and a resource. Figure 14 depicts this situation where a client end system 80 communicates with a resource system 81. The client end system comprises the client application 82, an e-speak core 83 including SLS layer 84, and a transport entity 85. Similarly, the resource system comprises a resource 86, an e-speak core 87 with SLS layer 88, and a transport entity 89. The e-speak cores are shown hatched for clarity. The functionality provided by the e-speak cores 83, 87 in addition to that provided by the SLS layer, is represented by respective services blocks, 90 and 91; this additional functionality includes resource registration, metadata export, and resource discovery. Once the client 82 has determined by reference to core services 90 that resource 86 can provide a desired service and is likely to allow the client access, the client can seek to establish a secure session with resource 86 using the services of the SLS layers 83 and 88 in the manner previously described,

The client and e-speak core may not, however, always reside on the same end system. For example, a client may simply be a small application 93 running in a web browser 94 (see Figure 15) with the most convenient e-speak core 95 being one hosted by a broker system 96 connected to the web. By providing browser 94 with an SLS XML plug-in, client 93 can

establish a secure session 97 over an HTTP connection with broker application 98 running on system 96 and make use of core 95 to locate a target resource 100. The client can then establish a nested session 101 with the resource 100, tunneling through the broker system; the connection between the broker system and the system running resource 100 is, for example a TCP connection. Of course, setting up the sessions 97 and 101 requires the participating parties to prove required attributes to each other in the manner already explained when describing the SLS protocol.

It may not always be acceptable to install an SLS plug-in into a web browser. In this case the client running in the browser could use an HTTP/SSL secure connection 105 (see Figure 16) to talk to the broker application 98 with the broker authenticating the client by any suitable method (client side certificates, basic authentication or digest authentication). The broker application 98 then creates a client proxy 106 to represent the client 93 in an SLS session 107 established using the e-speak core 95 on the broker system 96 to the resource 100. It will not be a secure end to end session between the web browser and the resource 100 as the proxy has to be trusted.

## REFERENCES

- [1] T. Dierke *et al.*, "The TLS Protocol, Version 1.0", IETF RFC2246 January 1999.
- [2] C. Ellison *et al.*, "SPKI Certificate Theory", IETF RFC2693 September 1999.
- [3] C. Ellison *et al.*, "SPKI Examples", IETF RFC 2692 September 1999
- [4] S. Kent *et al.*, "Security Architecture for the Internet Protocol" IETF RFC 2401 November 1998.
- [5] National Institute of Standards and Technology. *Secure Hash Standard*, April 17, 1995. FIPS PUB 180-1, available from <http://www.itl.nist.gov/fipspubs/>.
- [6] Mihir Bellare, Ran Canetti, and Hugo Krawczyk. "Keying hash functions for message authentication". In Neal Koblitz, editor, *Advances in Cryptology-CRYPTO '96*, volume 1109 of *Lecture Notes in Computer Science*, pages 1-15. Springer-Verlag, 1996.
- [7] National Institute of Standards and Technology. *Data Encryption Standard (DES)*, 1999. DRAFT FIPS PUB 46-3, available from <http://www.ncsl.nist.gov/fips/>.

- 10



## Appendix

This Appendix forms an integral part of the specification.

### 5 XML DTD for SLS

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<!-- A DTD (Document Type Definition) for the Session Layer Security Protocol (SLS).-->
```

```
<!DOCTYPE SLS [
```

```
<!-- XML SLS DTD -->
```

10

```
<!ENTITY % uint8 "#PCDATA" >
```

```
<!-- Where the PCData is an 8-bit unsigned integer. -->
```

```
<!ENTITY % uint32 "#PCDATA" >
```

15

```
<!-- Where the PCData is an 32-bit unsigned integer. -->
```

```
<!ENTITY % Address "#PCDATA" >
```

```
<!ENTITY String "#PCDATA" >
```

20

```
<!ELEMENT Opaque (#PCDATA) >
```

```
<!ATTLIST Opaque
```

```
encoding CDATA "urn:ietf-org:base64" >
```

```
25 <!ELEMENT % Object (#PCDATA) >
```

```
<!ATTLIST Object
```

```
encoding CDATA #IMPLIED >
```

```
<!ELEMENT % ADR (#PCDATA) >
```

```
30 <!-- Where the PCData conforms to the ADR syntax. -->
```

```
<!ATTLIST ADR
```

```
entity CDATA #IMPLIED >
```

<!-- Can use the entity to indicate which ADR syntax we want. -->

<!ELEMENT % ADRList (#PCDATA) >

<!-- Where the PCADATA conforms to the ADR syntax for a list. -->

5 <!ATTLIST ADR

entity CDATA #IMPLIED >

<!-- Can use the entity to indicate which ADR syntax we want. -->

10 <!ENTITY % MessageType " ALERT | HANDSHAKE | APPLICATION\_MESSAGE |  
TUNNEL " >

<!ENTITY % EncodingType " SECURE\_DATA | PROTECTED\_DATA | CLEAR\_DATA "  
>

<!ELEMENT versionMajor (%uint8;) >

15 <!ELEMENT versionMinor (%uint8;) >

<!ELEMENT spi (%uint32;) >

<!ELEMENT serial (%uint32;) >

<!ELEMENT inReplyTo (%uint32;) >

<!ELEMENT messageType (%MessageType;) >

20 <!ELEMENT encodingType (%EncodingType;) >

<!ELEMENT toAddress (%Address;) >

<!ELEMENT fromAddress (%Address;) >

<!ENTITY % PDUdata " SecureData | ProtectedData | ClearData " >

25 <!ELEMENT PDU (

versionMajor,

versionMinor,

spi,

serial,

30 inReplyTo?,

messageType,

encodingType,

002027" 524E260

5 <!ELEMENT SecureData ( Opaque ) >

<!ELEMENT mac ( Opaque )>

<!ELEMENT Alert ( level, code, detail? ) >

<!ELEMENT code ( %AlertCode; )>

&lt;!ENTITY % AlertLevel " FATAL | WARNING | DEBUG " &gt;

```
20      CLOSE_NOTIFY
      | UNEXPECTED_MESSAGE
      | BAD_SPI
```

```
25 | HANDSHAKE_FAILURE
   | BAD_CERTIFICATE
   | UNSUPPORTED_CERTIFICATE
   | CERTIFICATE_REVOKED
```

```
30 | ILLEGAL_PARAMETER
   | BAD_PROTOCOL_VERSION
   | INSUFFICIENT_SECURITY
```

| ERROR

" &gt;

```
| HandshakeStart
| HandshakeReply
| HandshakeFinish
">
```

10

```
versionMajor,  
versionMinor,  
spi,  
group,  
keyData,  
cipherSuiteList,  
tags?,  
query? ) >
```

15

20

```
<!ELEMENT group (%ADR;) >
<!ELEMENT keyData (%ADR;) >
<!ELEMENT cipherSuiteList (%ADRList;)>
<!ELEMENT tags (%ADRList;) >
<!ELEMENT query (%ADR;) >
```

25

30

```
<!ELEMENT HandshakeReply (
  versionMajor,
  versionMinor,
  spi,
  spiB,
```

**cipherSuite,**

proof?,

response?,

5 tags?,

relay?,

forwardAddress?,

signature )&gt;

```
10 <!ELEMENT spiB (%uint32;)>
```

&lt;!ELEMENT cipherSuite (%ADR;)&gt;

<!ELEMENT proof (%ADR;)>

<!ELEMENT response (%ADR;)>

&lt;!ELEMENT relay (%boolean;)&gt;

```
15    <!ELEMENT forwardAddress (%Address;)>
```

<!ELEMENT signature (%ADR;)>

&lt;!ENTITY % boolean " FALSE | TRUE " &gt;

20 &lt;!ELEMENT HandshakeFinish (

**spi,**

proof?,

signature)&gt;

25 <!ELEMENT ApplicationMessage ANY>

<!ELEMENT MessageForResource (

**versionMajor,**

versionMinor,

30      **primaryResource,**

**secondaryResources?**

tags?,

handlerInfo?,  
 payloadType,  
 payload? )>

5 <!ELEMENT primaryResource (ResourceReference)>  
 <!ELEMENT secondaryResources(ResourceReference\*)>  
 <!ELEMENT handlerInfo (ResourceInfo\*)>  
 <!ELEMENT payloadType (%PayloadType;)>

10 <!ELEMENT ResourceReference (#PCDATA) >  
 <!ELEMENT ResourceInfo (#PCDATA) >

<!ENTITY PayloadType "

METHOD\_CALL

15 | METHOD\_RESULT

| EXCEPTION

| EVENT

">

20 <!ELEMENT payload (MethodCall | Object )>

<!ELEMENT MethodCall (

interfaceName,

methodName,

25 args

)>

<!ELEMENT interfaceName (%String;)>

<!ELEMENT methodName (%String;)>

30 <!ELEMENT args (Object\*)>



00403F" 524EE260